LPI-Japan主催

Javaエンジニアのための HTML5プロフェッショナル認定試験 レベル2 解説無料セミナー

2016年1月23日

アシアル株式会社　生形可奈子

アシアルについて

アシアルは、「エンジニアリングでインターネットの成長を牽引する」という事業コンセプトのもと、HTML5・JavaScript等のWeb技術をベースとしたデベロッパー支援事業を行っています。

モバイルアプリ開発環境

UI/UX設計

システム構築・コンサル

セミナー・スクール

Asial
Engineering the future

自己紹介

生形　可奈子（うぶかた・かなこ）

- 講師・Monacaエバンジェリスト
- Java/C#を中心とした業務・制御系エンジニア出身
- 著書：「スラスラわかるJavaScript」
  「Monacaで学ぶはじめてのプログラミング
  ～モバイルアプリ入門編～」

目次

- HTML5プロフェッショナル認定試験とは
  - 概要
  - 試験範囲
- 頻出・難解ポイント解説
  - JavaScriptのデータ型
  - オブジェクト
  - 文字列／配列
  - 関数とスコープ
  - prototype／継承

HTML5プロフェッショナル認定試験

HTML5とは

- 2014年10月正式勧告
- リッチクライアント・アプリケーションのプラットフォーム
- マルチデバイス対応・マルチメディア対応
- 広義ではCSS3やJavaScriptによる3Dグラフィック、WebSocket、デバイスアクセス、クライアントストレージ等も含む

HTML5プロフェッショナル認定試験とは

- 特定非営利活動法人LPI-Japanが実施する、HTML5および周辺技術の知識レベルを測る認定制度です。

- 試験の難易度を示す2種類のレベルがあり、段階的に受験します。
  - Level1

    マルチデバイスに対応した静的なWebコンテンツを HTML5を使ってデザイン、作成できるレベル

  - Level2

    システム間連携や最新のマルチメディア術に対応したWebアプリケーションや動的Webコンテンツの開発・設計ができるレベル

Level2の出題範囲 （重要度の高いもの）

出題範囲	重要度
JavaScript	
JavaScript文法	★★★★★★★★★★ 10
WebブラウザにおけるJavaScript API	
イベント	★★★★★★★★ 8
ドキュメントオブジェクト／DOM	★★★★★★ 6
ウィンドウオブジェクト	★★★★★★★★ 8
グラフィックス	
Canvas(2D)	★★★★★★ 6

受験について

- **試験方式はコンピュータベーストテスト（CBT）です。試験配信会社の「ピアソンVUE」を通して受験します。**

問題数	40～45問
試験時間	90分
合格ライン	約7割
回答方式	殆どが選択式（複数回答あり） 記述式も１問程度 コードリーディング問題が多い
受験料	¥15,000（税抜）

JavaScriptについて

JavaScriptの歴史

JavaScriptのはじまり

- 当初はMocha、LiveScriptという名前だった
- 開発元のNetscape社とSun社（現Oracle）の提携により、JavaScriptに改名される
- 初期はWebページに簡単な動きを付けたり、フォーム入力を補助することが目的の簡易的言語だった

モダンJavaScriptの登場

JavaScript最新動向

- Gmailなどのクラウドサービスの登場により、ブラウザ上で実現できることが増え、JavaScriptが脚光を浴びるようになる
- 様々なJavaScript製フレームワークが台頭してきたことにより、大規模なWebアプリケーションの構築が可能となった
- 進化系として、サーバサイドプログラムの構築やスマートフォンアプリの制作にも用いられるケースが増えてきた

EcmaScript6

JavaScriptの標準仕様

- JavaScriptは、各ブラウザに独自に実装されており、それぞれ 独自仕様が含まれている場合がある
- ActionScriptやJScript等、JavaScript以外の言語も存在する
- 上記言語を標準化するために作られた仕様が「EcmaScript」
- 言語仕様のみを定義しており、HTML5 APIやDOMは含まれない

最新バージョンは6

- 2015年6月17日に承認

JavaとJavaScript

JavaScriptが影響を受けている言語

- コードの記述方法はJavaに似ている
- 言語仕様として最も影響を受けているのは、SchemeとSelf

JavaとJavaScriptの主な違い

言語	型システム	オブジェクト指向分類
Java	静的型付け	クラスベース
JavaScript	動的型付け	プロトタイプベース

JavaScriptのデータ型

JavaScriptのデータ型

JavaScriptは動的型付け言語

- データの型は実行時に決定されるため、変数宣言時に型を限定しない

変数宣言の例

```
var 変数名;
```

実行時にデータ型が判定される

```
console.log(1 + 1);          // 2
console.log(1 + '1');        // 11
```

JavaScriptのデータ型

データ型一覧

型の分類	代表的な型
プリミティブ型 （１つの値のみを持つ）	数値、文字列、論理値 null（値が存在しない） undefined（値が定義されていない）
オブジェクト型 （プロパティの集合体）	オブジェクト、関数、配列など

nullとundefined

- **null … 値が存在しないことを意図的に示す値**
  - getElementById()で取得しようとする要素がHTML文書内に存在しない場合などに、メソッドの戻り値として返却される
- **undefined … 値がそもそも定義されていないことを示す値**
  - 初期化されていない変数、定義されていないプロパティ、return文のない関数の戻り値などはすべてundefinedで表される

等価演算子

- JavaScriptの等価演算子には二種類ある
  - 値が等しいかどうかのみを比較する緩やかな等価演算子
  - データの型まで含めて等しいかどうかを比較する厳密な等価演算子

演算子	説明	例	結果
==	緩やかな等価演算子	5 == '5'	true
===	厳密な等価演算子	5 === '5'	false
!=	緩やかな不等価演算子	5 != '5'	false
!==	厳密な不等価演算子	5 !== '5'	true

条件式の論理値判定

条件式における値の評価

- 論理値以外の値をif文などの条件式として指定すると、以下のように判定される

true	false
0以外の数値 1文字以上の文字列 オブジェクト（中身が空でも）	0 空文字（""） undefined null

条件式の記述例

- 引数の省略チェックなどに用いられる

```
function check(x) {
    // 引数xが空だったらエラーにする
    if(!x) {
        console.log("エラー：値が入っていません");
        return;
    }
    // 以降、省略
}
```

条件式の記述に関する注意点

- 0や空文字もfalseとして判定されることに注意

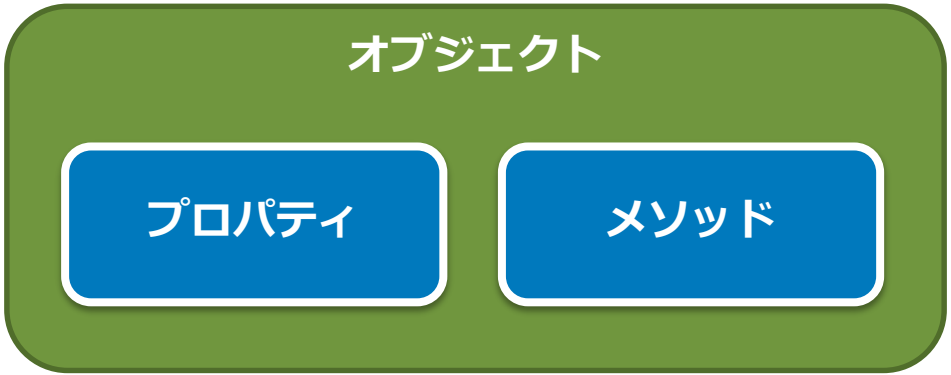
```
function check(x) {
    // 引数xが空だったらエラーにする
    if(!x) {
        console.log("エラー：値が入っていません");
        return;
    }
    // 以降、省略
}

check();        // 想定通りのエラー
check(0);       // 想定外のエラー
```

オブジェクトの基本

オブジェクトの基本

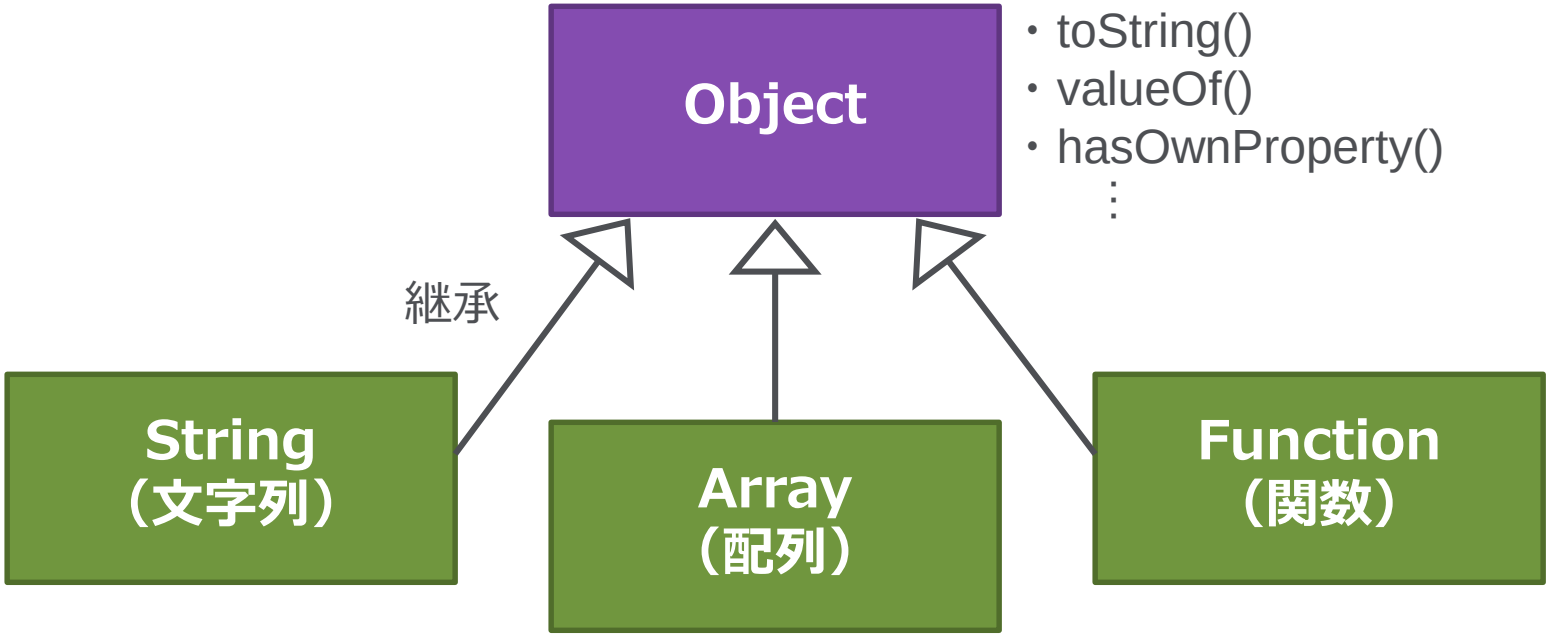
プロパティとメソッド

- オブジェクトは、複数のデータの集合体である
- オブジェクト内のデータには、オブジェクトの属性値を保持する「プロパティ」と、何らかの操作を行う「メソッド」がある



オブジェクトの種類

- Objectオブジェクトはすべてのオブジェクトの親となっている



オブジェクトの生成

オブジェクトの生成方法

```
var 変数名 = new オブジェクトの型名(引数);
```

- Objectオブジェクトの生成は、オブジェクトリテラル{}で代用することが可能
- Arrayブジェクトの生成は、配列リテラル[]で代用することが可能

オブジェクト生成の例

```
var list = new Array(1,2,3);
```

オブジェクト生成の仕組み

オブジェクト生成時の流れ

```
var list = new Array(1,2,3);
```

① オブジェクトの生成時に、コンストラクタという関数が実行される

② コンストラクタは、新しくオブジェクトを生成し、それを戻り値として返却する（このオブジェクトをインスタンスと呼ぶ）

typeof演算子

typeof演算子

- データの型を調べる演算子なので、StringやArrayなどのオブジェクトはほぼすべて「object」と判定される

  ※ 関数は「function」と判定される

```
var list = new Array(1,2,3);
console.log(typeof list);  // object
```

instanceof演算子

instanceof演算子

- どのコンストラクタによって生成されたインスタンスなのかを判定するので、オブジェクトの種類を調べることができる

```
var list = new Array(1,2,3);
console.log(list instanceof Array);  // true
```

ユーザー定義オブジェクト

ユーザー定義オブジェクト

- ユーザー定義オブジェクトの作成
  - newを使用

```
var オブジェクト名 = new Object();
```

  - オブジェクトリテラルを使用

```
var オブジェクト名 = {};
```

ユーザー定義オブジェクト

- プロパティへのアクセス
  - ドットを使用

```
変数名.プロパティ名
```

  - 角かっこを使用

```
変数名["プロパティ名"]
```

オブジェクトの例

- オブジェクトの生成とプロパティへのアクセス

```
var obj = {
    name: "生形"
};

// プロパティへのアクセス
console.log(obj.name);
// 以下のように記述することも可能
var propertyName = "name"
console.log(obj[propertyName]);
```

文字列

JavaScriptの文字列

- 文字列リテラルは、以下の二種類
  - シングルクォート・ダブルクォートのどちらでも可

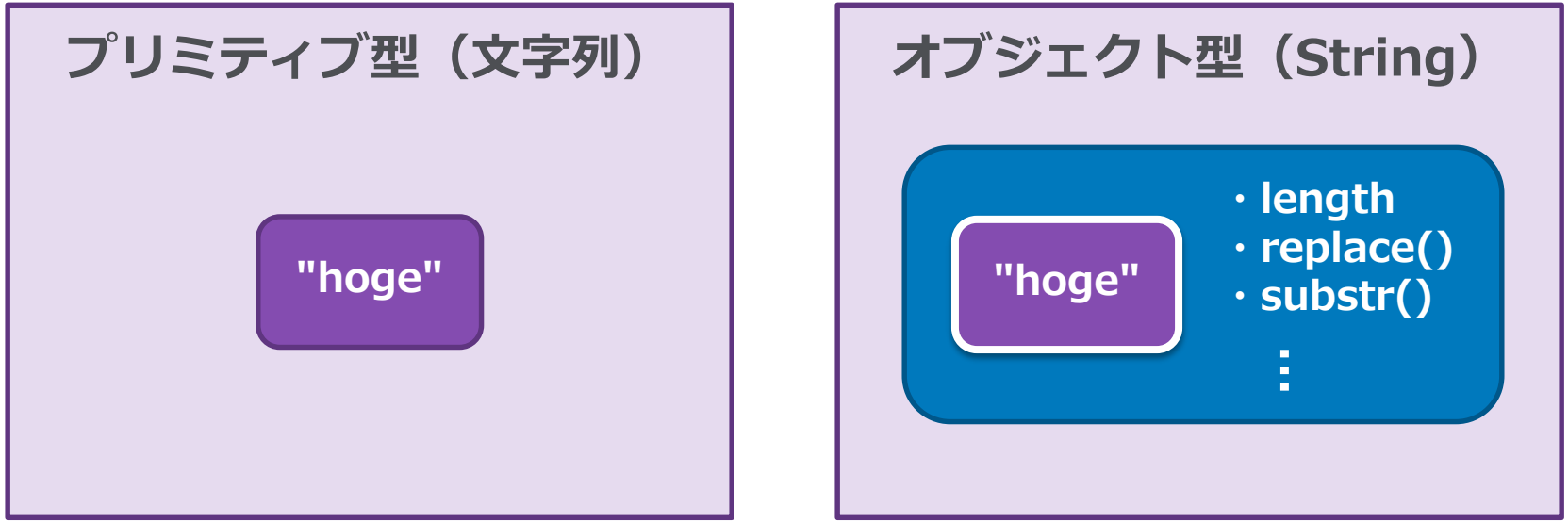
```
'こんにちは'
"こんにちは"
```

- 文字列にはプリミティブ型とオブジェクト型の二種類がある
  - 文字列リテラルはプリミティブ型となる

```
var str1 = "こんにちは";                  // プリミティブ型
var str2 = new String("こんにちは");  // オブジェクト型
```

プリミティブ型とオブジェクト型

- プリミティブ型は１つの値だけ、オブジェクト型は複数のメソッドやプロパティを持つ



ラッパーオブジェクト

- ラッパーオブジェクト … プリミティブ型に対応するオブジェクト

プリミティブ型	ラッパーオブジェクト
数値	Number
文字列	String
論理値	Boolean

問題：プリミティブ型に対してプロパティを指定

- 以下の実行結果は？

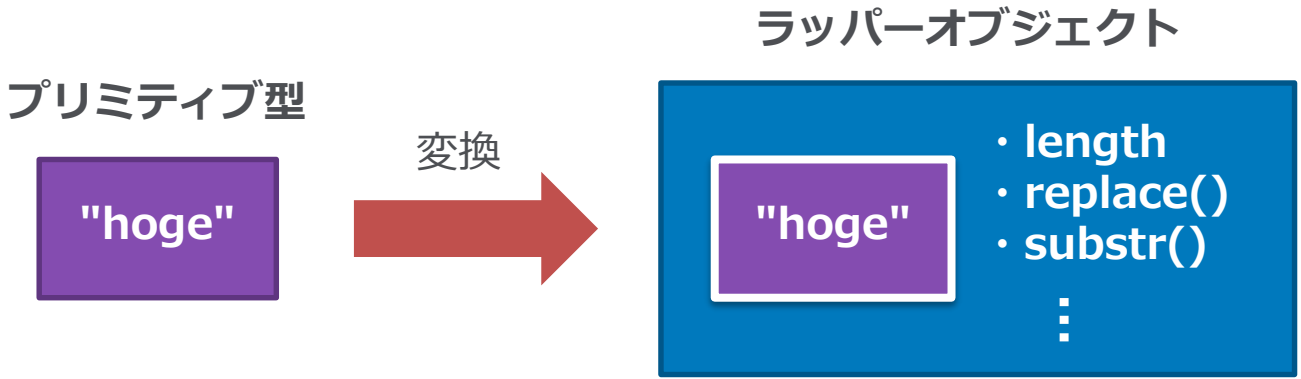
```
console.log("hoge".length);
```

A) 4
B) null
C) undefined
D) エラー

ラッパーオブジェクトへの暗黙的変換

- プロパティを呼び出した場合、プリミティブ値から対応するラッパーオブジェクトへの暗黙的な変換が行われる

```
"hoge".length
```



問題：プリミティブ型とラッパーオブジェクトの比較

- 以下の実行結果は？（true / false）

```
var a = "hoge";
var b = new String("hoge");
console.log(a == b);
```

- 以下の実行結果は？（true / false）

```
var a = "hoge";
var b = new String("hoge");
console.log(a === b);
```

問題：値の比較、参照の比較

- 以下の実行結果は？（true / false）

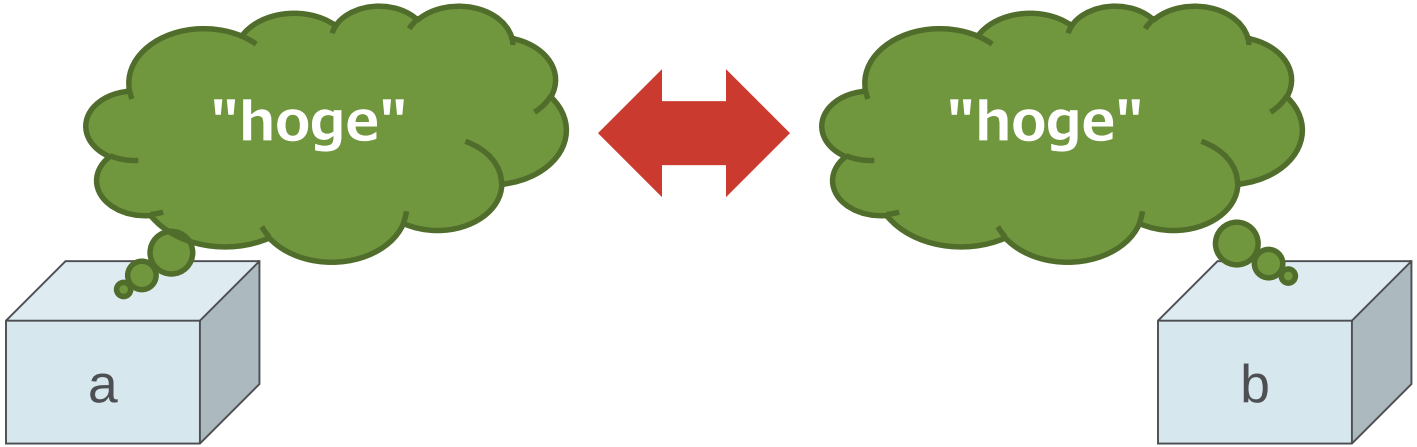
```
var a = "hoge";
var b = "hoge";
console.log(a == b);
```

- 以下の実行結果は？（true / false）

```
var a = new String("hoge");
var b = new String("hoge");
console.log(a == b);
```

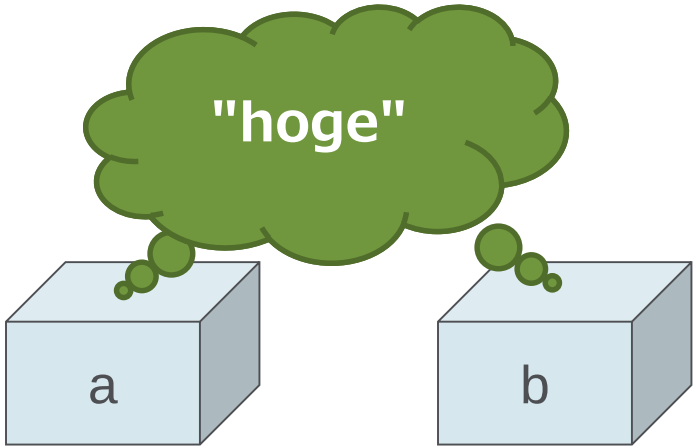
値の比較、参照の比較

- プリミティブ型は値そのものを比較する
- オブジェクト型は参照しているオブジェクトが同じものかどうかを比較する



同じオブジェクトを参照している例

```
var a = new String("hoge");
var b = a;
console.log(a == b);
```



この場合は同じオブジェクトを
参照しているのでtrue

配列

配列の基本文法

- 配列の作成
  - newを使用

```
var 配列名 = new Array(1, 2, 3);
```

  - 配列リテラルを使用

```
var 配列名 = [1, 2, 3];
```

JavaScriptの配列の特徴

- 配列の要素数は柔軟に変更することが可能

```
var list = [];     // 空の配列を作成
list[0] = "AAA";
list[1] = "BBB";
list[2] = "CCC";
```

問題：配列のインデックス

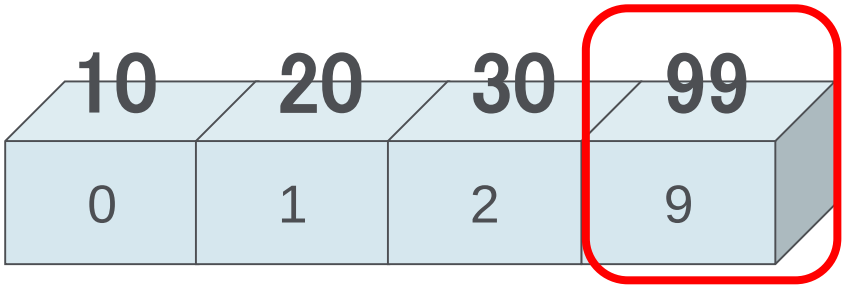
- 以下の実行結果は？

```
var array = [10, 20, 30];
array[9] = 99;
console.log(array.length);
```

A) 3
B) 4
C) 10
D) undefined

配列のインデックス

- インデックスは連番でなくても割り振ることができる



- lengthプロパティは最大インデックス+1を返す

関数

JavaScriptにおける関数

- 関数もオブジェクトの一種
  - 関数はFunctionという名前のオブジェクトである
  - オブジェクトの一種なので、変数の値として代入することが可能。
    オブジェクトのメソッドは、Fuctionオブジェクトをプロパティに代入することで実現している

```
var obj = {
    method1: function (message) {
        alert(message);
    }
};
```

関数の定義いろいろ①

関数宣言

```
func("hello");
function func(message) {
        alert(message);
}
```

- 関数にfuncという名前をつけている
- 関数は静的に作成される（スクリプト実行前に評価される）

関数の定義いろいろ②

関数式

```
var func = function(message) {
        alert(message);
};
func("hello");
```

- funcという変数に関数を代入している
- この場合、関数名は省略できる（無名関数や匿名関数と呼ばれる）

関数の定義いろいろ③

即時関数

```
(function(message) {
        alert(message);
})("hello");
```

- (関数の定義)(引数); とすることで定義と呼び出しを同時に行う
- 主にスコープを限定する用途などで利用される

EcmaScript6から追加される機能

- デフォルト引数
- 可変長引数
- アロー記法

デフォルト引数

■デフォルト引数

```
function f(x, y=20){
    console.log(x);  // 10
    console.log(y);  // 20
}

f(10);
```

- 引数に値が渡されなかった場合に代入する値を指定することができる

可変長引数

可変長引数

```
function f(...args) {
      for(var i=0; i<args.length; i++) {
            console.log(args[i]);
      }
}

f(10, 20);
f(10, 20, 30, 40);
```

- 引数の数を動的に変更することができる

アロー記法

従来の記法

```
var f = function(x) { return x * 2; };
```

アロー記法

```
var f = x => x * 2;
```

- functionキーワードを省略できる
- 引数が１つの場合は、()を省略できる
- 処理がreturn文のみの場合は、{}とreturnキーワードを省略できる

変数のスコープ

スコープとは

- 変数はスコープ（有効範囲）内でのみ参照できる

```
var a = 10;  // aはグローバル変数なのでどこからでも参照可
function func1() {
    var b = 20;
    function func2() {
        var c = 30;
    }
}
```

cのスコープ

bのスコープ

- スコープは関数単位で作られる

スコープとは

- スコープ外で変数を参照した場合、ReferenceErrorが発生する

```
var a = 10;
function func1() {
    var b = 20;
    function func2() {
        var c = 30;
    }
}
func1();
console.log(a);  // 10
console.log(b);  // ReferenceError
```

スコープとは

- varをつけていない場合、その変数はグローバルスコープとなる

```
var a = 10;
function func1() {
    b = 20;
    function func2() {
        var c = 30;
    }
}
func1();
console.log(a);  // 10
console.log(b);  // 20
```

varを付け忘れているので、
aと同じグローバル変数になる

use strict

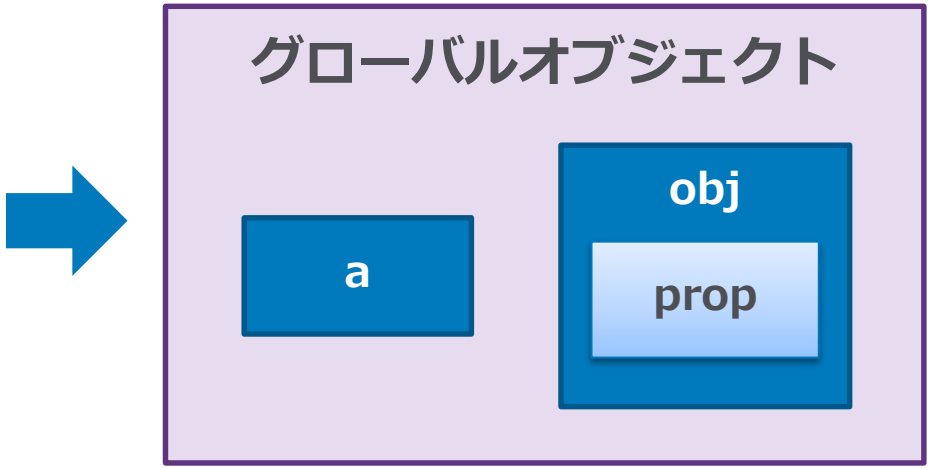
- use strict（厳格モード）を宣言すると、varの付け忘れなどの、推奨されない記述に対して警告が表示される

```
function func() {
    "use strict";
    b = 20;  // エラー
}
```

グローバル変数とは何か？

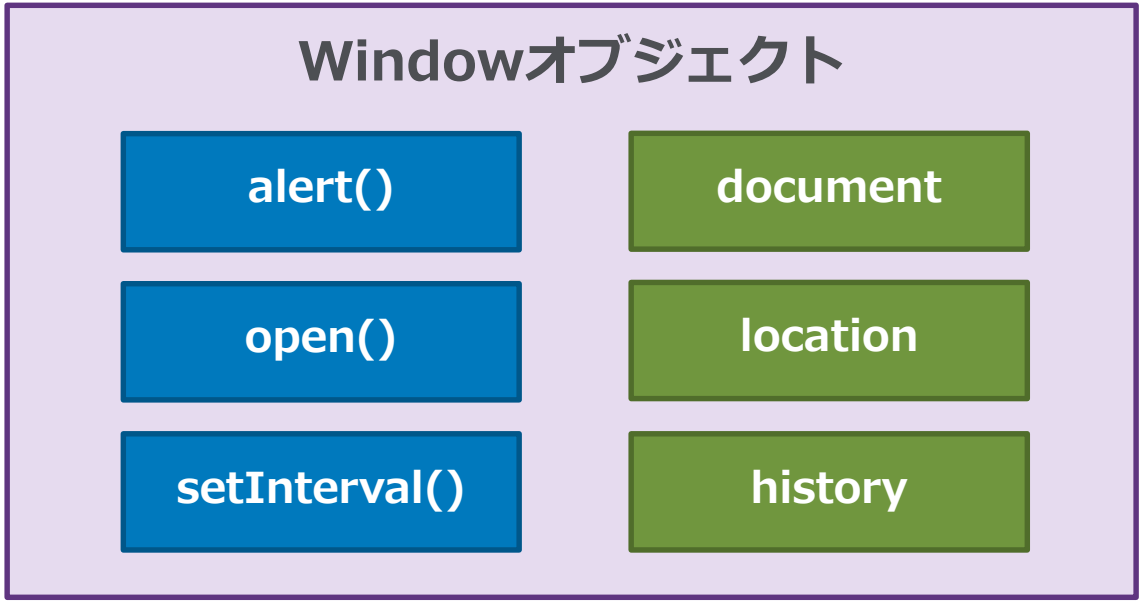
- スクリプト内の値やオブジェクトは全て、暗黙的に一番外側に存在しているグローバルオブジェクトに含まれている

```
<script>
    var a = 10;
    var obj = {
        prop: "hoge"
    };
</script>
```



グローバルオブジェクト

- ブラウザ環境におけるグローバルオブジェクトは、「Windowオブジェクト」である



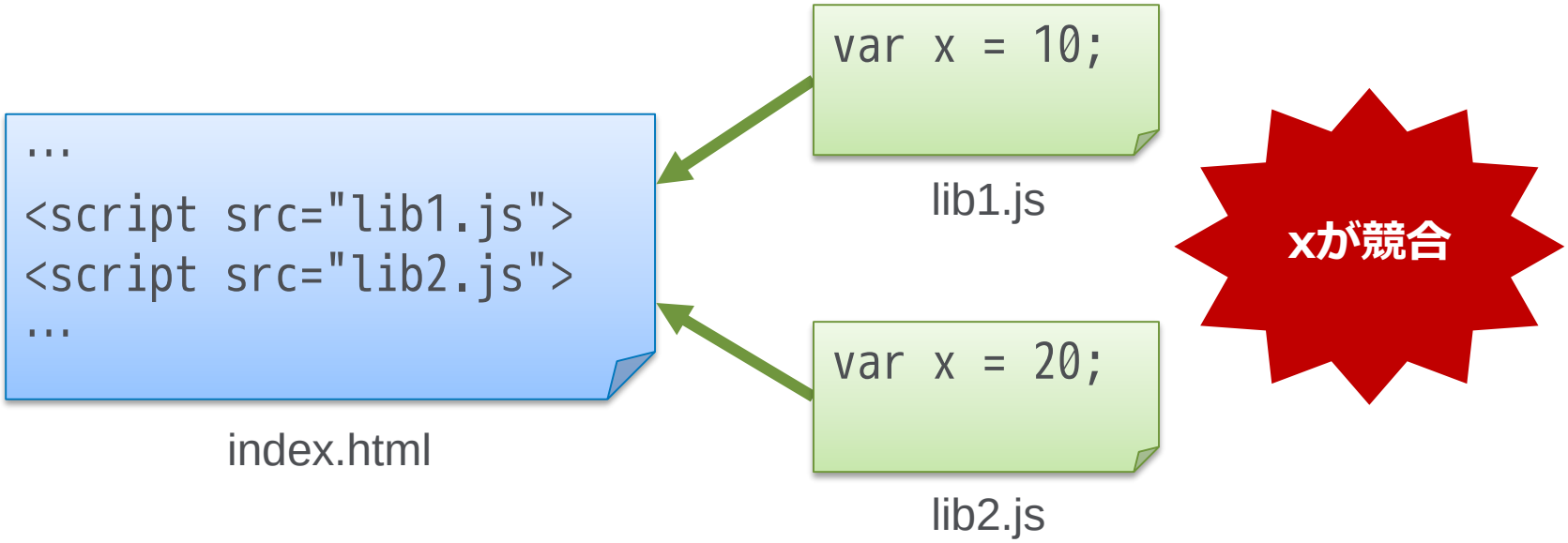
グローバル変数の正体

- グローバル変数＝グローバルオブジェクトのプロパティ

```
<script>
var a = 10;

console.log(a);          // 10
console.log(window.a);   // 10
</script>
```

グローバル変数利用の注意

- 複数のJSファイルを読み込んでいる場合に、グローバル変数が競合してしまう場合があるので利用は必要最小限に



シャドーイング

- シャドーイング（スコープの重複）は許可されている

```
var x = 10;  // グローバルスコープ
function func1() {
    var x = 20;  // ローカルスコープ
    console.log(x);  // 20
}
```

  - 上記の場合、グローバルスコープとローカルスコープに同名の変数が宣言されている。こういった場合はローカルスコープの変数が優先される

問題：JavaScriptのスコープ①

- 以下の実行結果は？

```
for(var i = 0; i < 3; i++) {
    var a = "hoge";
}
console.log(a);
```

A) エラー
B) undefined
C) null
D) "hoge"

JavaScriptのスコープ①

- JavaScriptにはブロックスコープがない
  - スコープが発生するのは関数内のみ
  - ただし、ES6ではブロックスコープを有効にするletキーワードが追加されている

```
if(true) {
    let a = "hoge";
}

console.log(a);  // a is not defined
```

問題：JavaScriptのスコープ②

- 以下の実行結果は？

```
var a = "global";
function func() {
    console.log(a);
    var a = "func";
}
func();
```

A) エラー
B) undefined
C) "global"
D) "func"

JavaScriptのスコープ②

ホイスティング（変数の巻き上げ）とは

- 関数の途中で宣言された変数は、関数の先頭に巻き上げて宣言される
- ただし、初期値として代入されている値は巻き上げない
- 先ほどのコードは以下のように解釈される

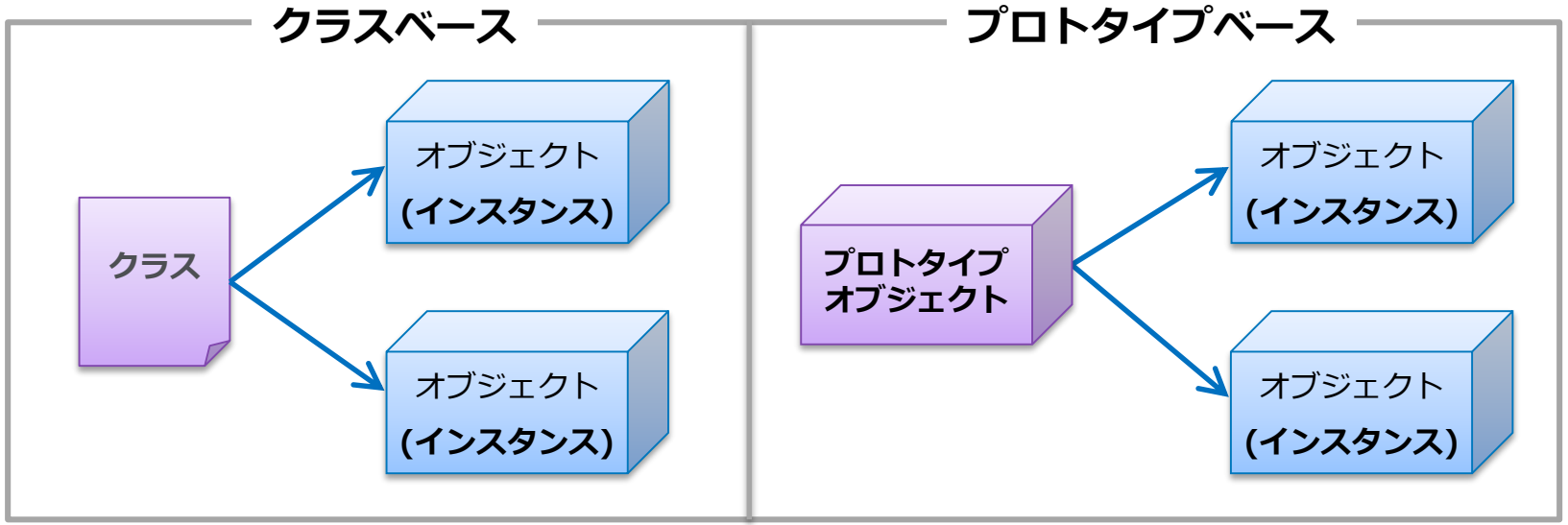
```
var a = "global";
function func() {
    var a;
    console.log(a);
    a = "func";
}
func();
```

変数宣言は、必ずスコープの先頭で行うこと

prototype

プロトタイプベースオブジェクト指向

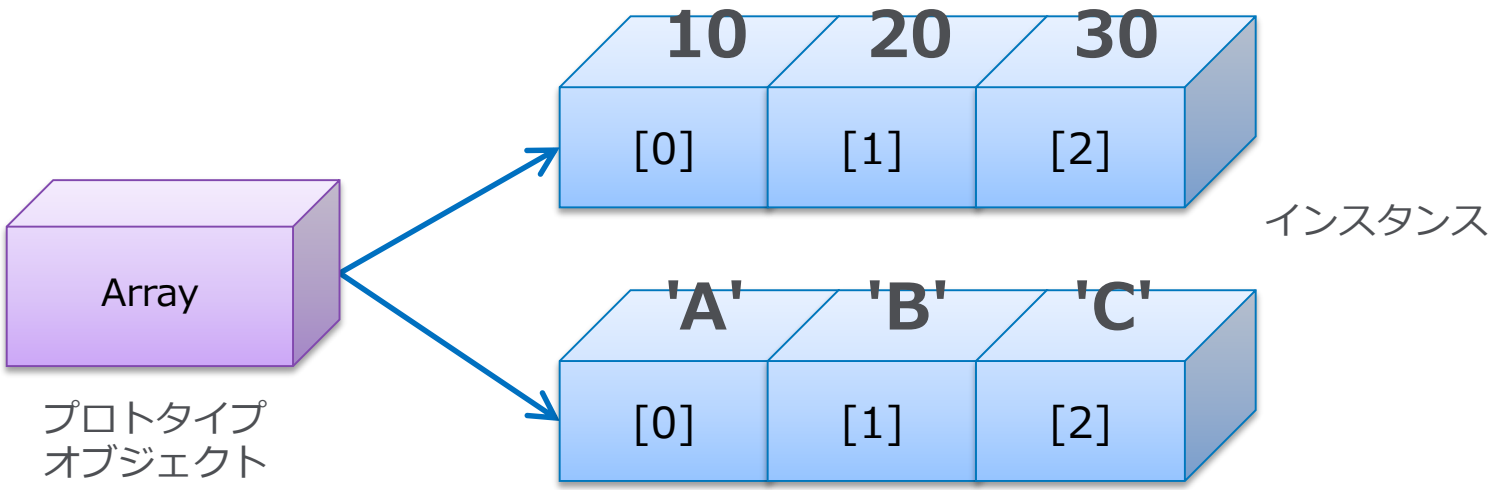
- JavaScriptはプロトタイプベースのオブジェクト指向言語
  - あるオブジェクトを基礎として、新しいオブジェクトを生成する仕組み
  - 新しく作られたオブジェクトは、「インスタンス」と呼ばれる



プロトタイプベースオブジェクト指向

- 組み込みオブジェクトもプロトタイプベースで作られている

```
var list1 = new Array(10, 20, 30);
var list2 = new Array('A', 'B', 'C');
```



プロトタイプベース指向に則ったオブジェクト定義

- プロトタイプオブジェクト＝関数
  - 基礎となるオブジェクト（クラスベース言語におけるクラスの役割）は、関数オブジェクトである

```
var Dog = function() { };
```

  - プロトタイプオブジェクトを元にインスタンスを生成するには、new 演算子を使う

```
var dog1 = new Dog();
```

プロトタイプベース指向に則ったオブジェクト定義

- コンストラクタ＝関数の処理
  - 関数内の処理はインスタンスを生成（new）した時に、コンストラクタとして実行される

```
var Dog = function(_name) {
    this.name = _name;
    this.show = function() {
        alert(this.name);
    };
};

var dog1 = new Dog("ポチ");  // コンストラクタを実行
```

this = インスタンス

プロトタイプベース指向に則ったオブジェクト定義

- インスタンス＝Objectオブジェクト
  - プロトタイプオブジェクトはFunction型だが、インスタンスはObject型となる

```
var Dog = function(_name) {
    // 省略
};
var dog1 = new Dog("ポチ");

console.log(typeof Dog);   // function
console.log(typeof dog1);  // object
```

プロトタイプベース指向に則ったオブジェクト定義

コンストラクタの暗黙的処理

- インスタンスがObject型になるのは、コンストラクタが暗黙的に新しいObjectを生成しているためである。

ソースコード

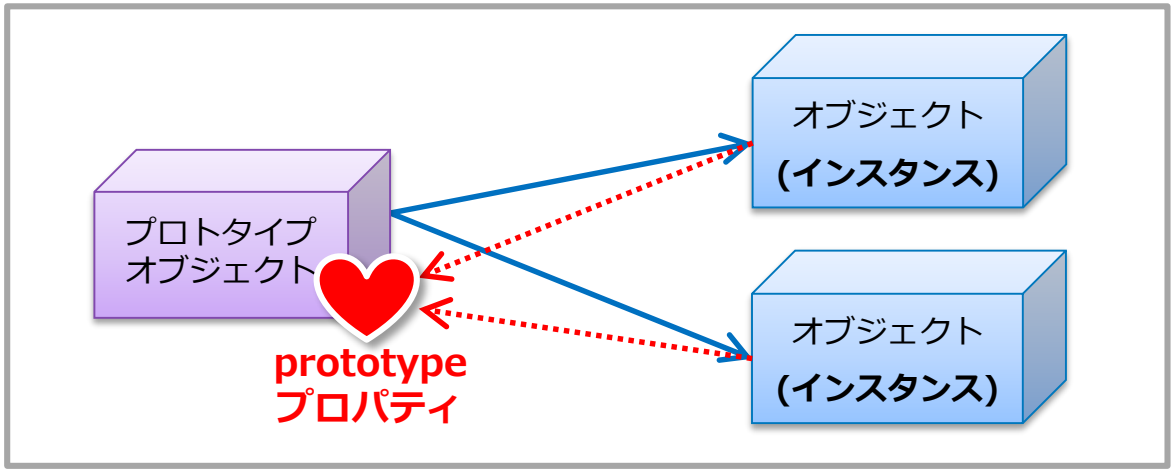
```
var Dog = function(_name) {
  this.name = _name;
  this.show = function() {
    alert(this.name);
  };
};
```

暗黙的処理

```
var Dog = function(_name) {
  var this = {};
  this.name = _name;
  this.show = function() {
    alert(this.name);
  };
  return this;
};
```

prototypeプロパティとは

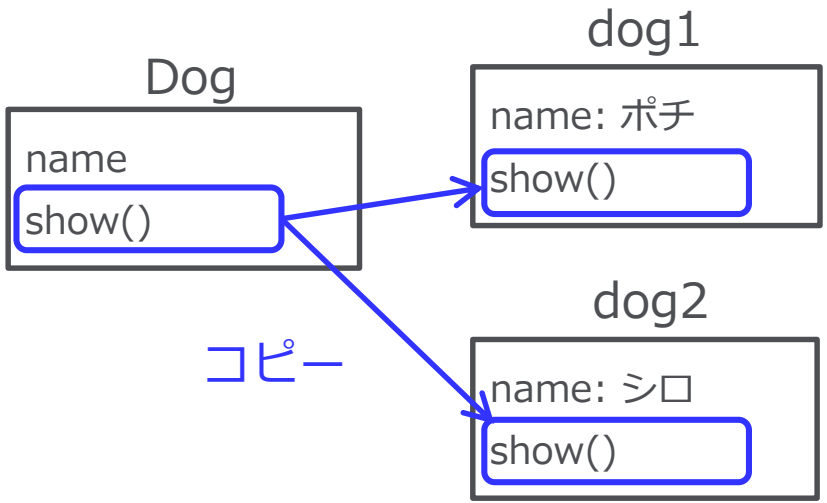
- オブジェクトのコア
  - プロトタイプオブジェクトが持つプロパティ
  - 各インスタンスは、プロトタイプオブジェクトのprototypeを参照する
  - prototypeプロパティには、各インスタンスで共通の機能を格納する



共通化されていない例

- オブジェクトをインスタンス化すると、元のオブジェクトで定義されたプロパティとメソッドが各インスタンスにコピーされる

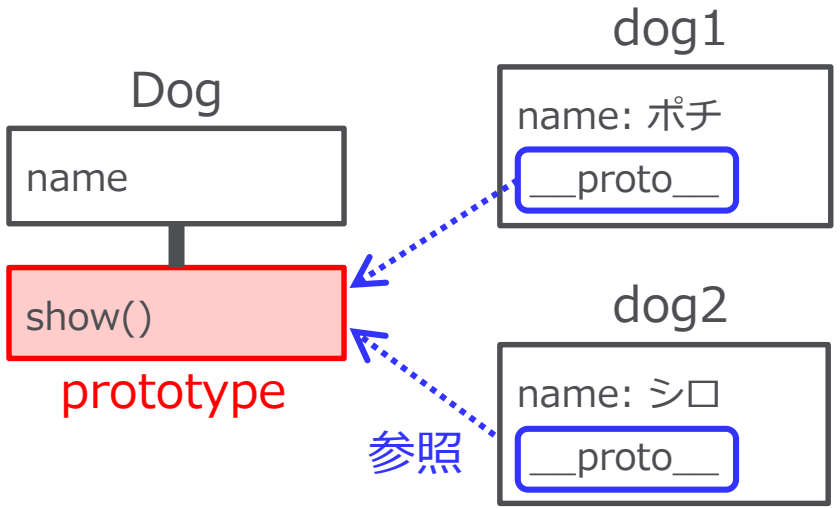
```
var Dog = function(_name) {
    this.name = _name;
    this.show = function() {
        alert(this.name);
    };
};
var dog1 = new Dog("ポチ");
var dog2 = new Dog("シロ");
```



共通化されている例

- 各インスタンスが内部的に持つ__proto__プロパティから元のオブジェクトのprototypeプロパティを参照する

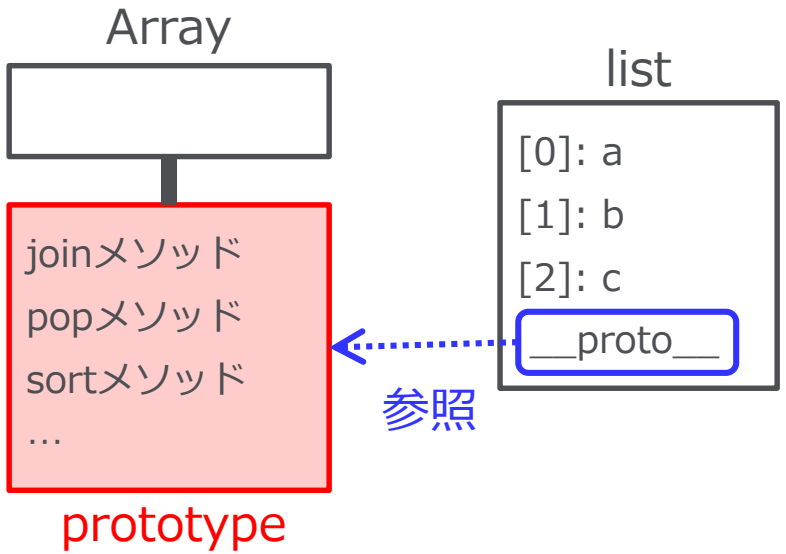
```
var Dog = function(_name) {
    this.name = _name;
};
Dog.prototype.show =
function() {
    alert(this.name);
};
var dog1 = new Dog("ポチ");
var dog2 = new Dog("シロ");
```



組み込みオブジェクトの例

- 以下は、Arrayのインスタンスからprototypeに定義されているjoinメソッドを呼び出している例

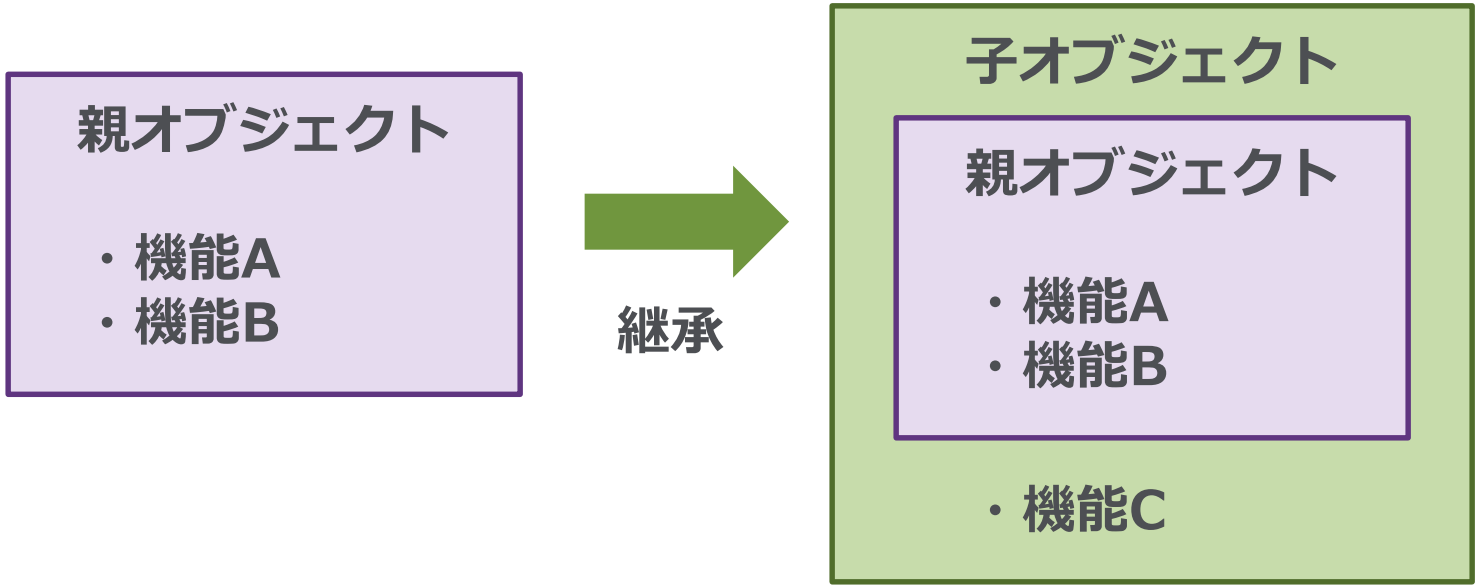
```
var list = ['a', 'b', 'c'];
var str = list.join('+');
console.log(str);  //a+b+c
```



継承

継承

- 継承とは
  - あるオブジェクトを拡張して、別のオブジェクトを作る仕組み



JavaScriptにおける継承の実現

最も単純な継承関係

- 子のprototypeプロパティに、親のインスタンスを代入する

```
// 親
var Parent = function() {
    this.x = 10;
};
Parent.prototype.show =
function() { alert(this.x); };

// 子
var Child = function() {
    this.y = 20;
};
```

```
// 継承
Child.prototype = new Parent();

var child = new Child();
child.show();
```

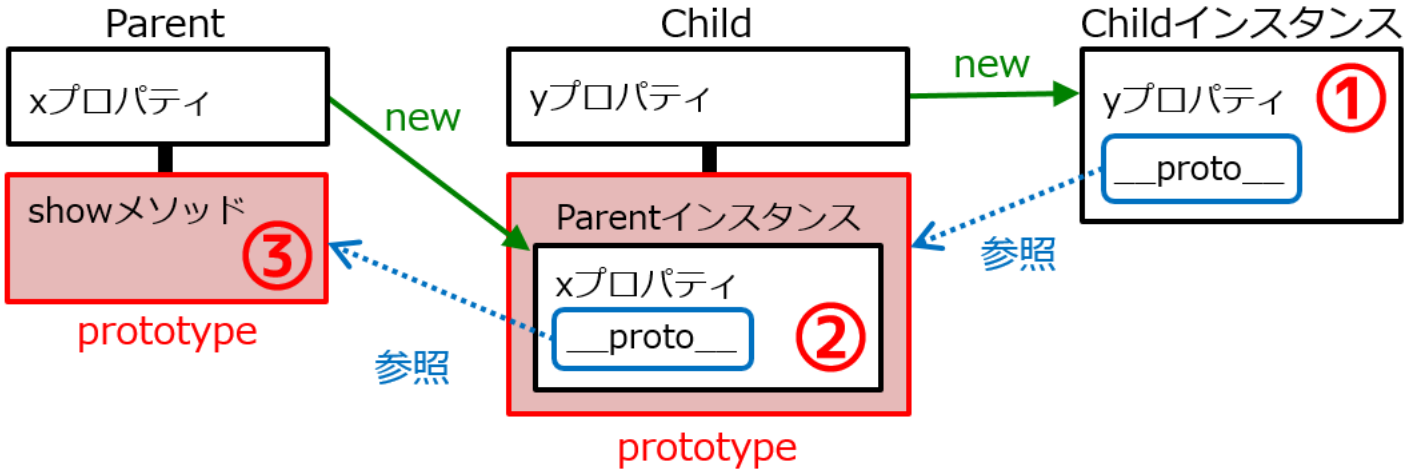
プロトタイプチェーン

- **プロパティを探索する順序**

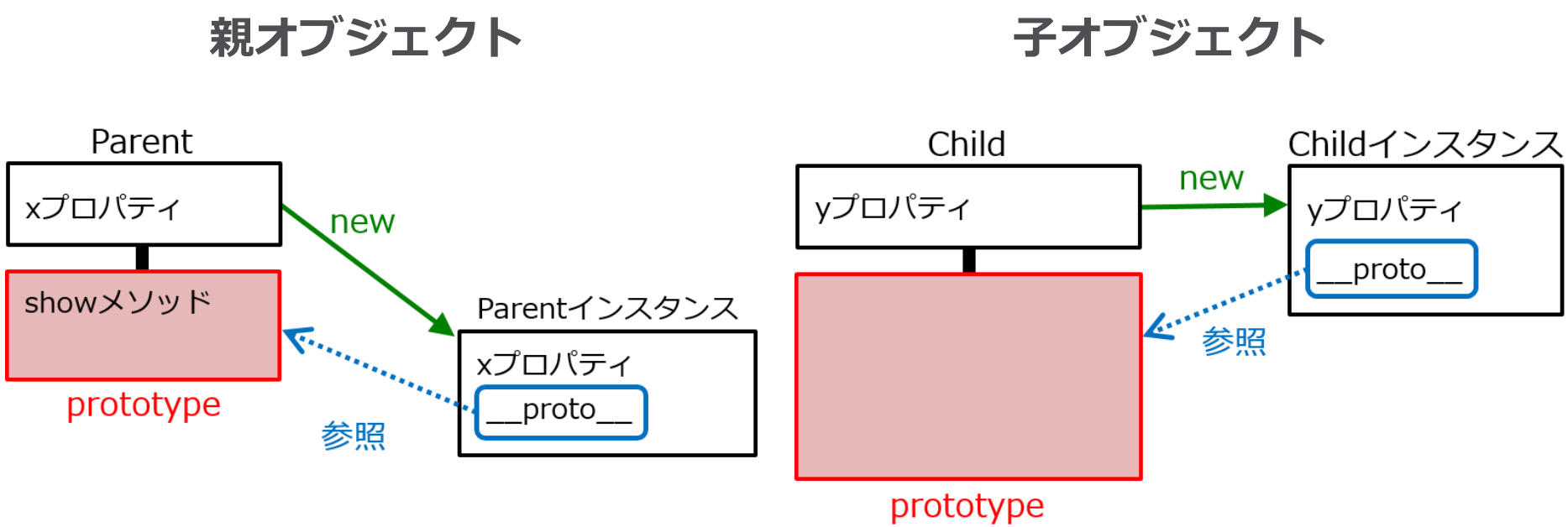
  ①自身のインスタンス内に該当プロパティがあるかどうかを調べる

  ②プロトタイプオブジェクトのprototype内を調べる

  ③親オブジェクトのprototype内を調べる



プロトタイプチェーン



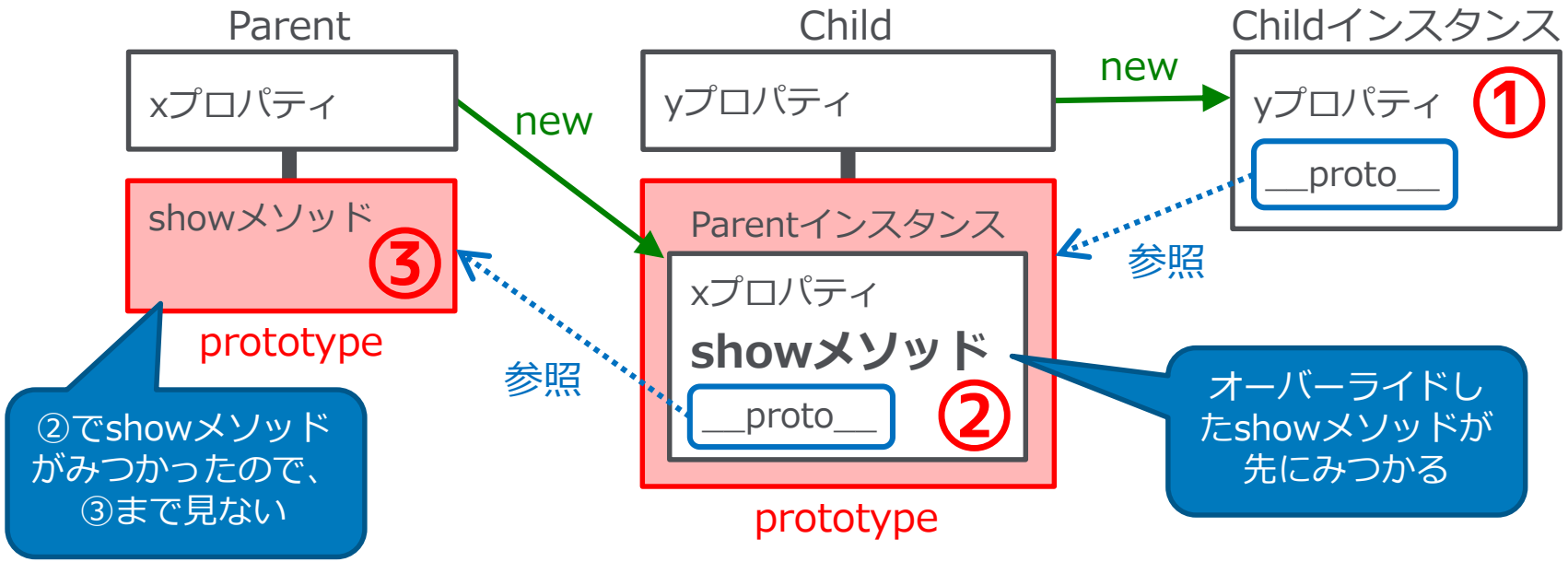
メソッドのオーバーライド

- 子のprototypeプロパティに、親が持つメソッドと同名のメソッドを定義する

```
// 親
var Parent = function() {
    this.x = 10;
};
Parent.prototype.show =
function() { alert(this.x); };

// 子
var Child = function() {
    this.y = 20;
};
```

```
// 継承
Child.prototype = new Parent();

// メソッドのオーバーライド
Child.prototype.show =
function() {
  alert(this.x + this.y);
};

var child = new Child();
child.show();
```

プロトタイプチェーンを利用したオーバーライドの実現



プロパティの走査

- for/in文
  - すべてのユーザ定義プロパティを走査する
  - プロトタイプチェーンを遡って走査される

```
var child = new Child();

for(var key in child){
    console.log(key + ':' + child[key]);
}
```

自身が持つプロパティのみを取得する

- hasOwnProperty
  - 自身が持つプロパティかどうかを調べ、論理値を返す

```
var child = new Child();

for(var key in child){
    if(child.hasOwnProperty(key)) {
        console.log(key + ':' + child[key]);
    }
}
```

ES6：class構文

- class構文により、クラスベース言語のように扱うことができる

```
class Parent {
    show(text) {
      console.log(text);
    }
};

class Child extends Parent {
    show() {
      super.show("child");
    }
};
```

メソッドはprototype内に定義される

キーワード１つで継承関係を実現

親オブジェクトの参照も容易

受験対策

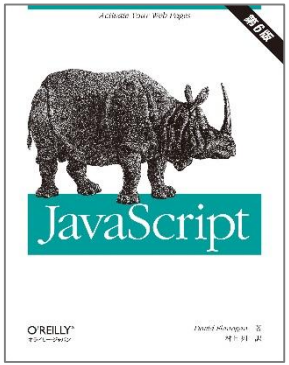
参考書籍

レベル２対策本

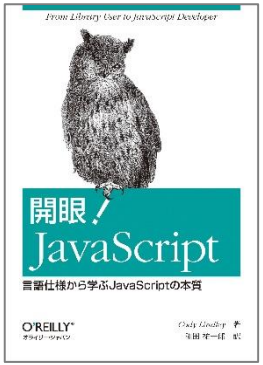
HTML5プロフェッショナル
認定試験レベル2攻略テキスト

コア・JavaScript

JavaScript



開眼！JavaScript



HTML5 API

HTML5 Web標準API バイブル

EcmaScript6の実習

https://babeljs.io/

LPI-JAPAN HTML5 Professional Certification

Open the Future with HTML5.